22 | Catalyst物理计划：你的SQL语句是怎么被优化的（下）？

2021-05-03 吴磊

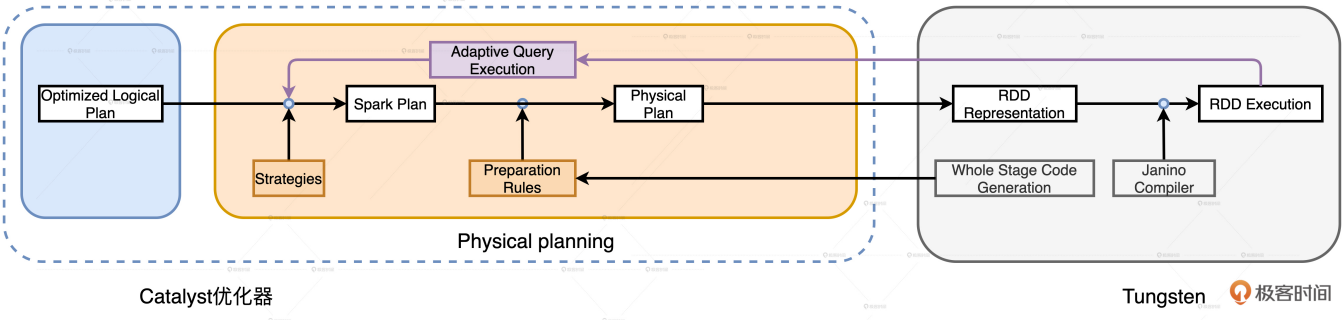
Spark性能调优实战 进入课程 >

讲述：吴磊
时长 17:30 大小 16.03M

你好，我是吴磊。

上一讲我们说了，Catalyst 优化器的逻辑优化过程包含两个环节：逻辑计划解析和逻辑计划优化。逻辑优化的最终目的就是要把 Unresolved Logical Plan 从次优的 Analyzed Logical Plan 最终变身为执行高效的 Optimized Logical Plan。

但是，逻辑优化的每一步仅仅是从逻辑上表明 Spark SQL 需要“做什么”，并没有从执行层面说明具体该“怎么做”。因此，为了把逻辑计划交付执行，Catalyst 还需要把 Optimized Logical Plan 转换为物理计划。物理计划比逻辑计划更具体，它明确交代了 Spark SQL 的每一步具体该怎么执行。



物理计划阶段

今天这一讲，我们继续追随小 Q 的脚步，看看它经过 Catalyst 的物理优化阶段之后，还会发生哪些变化。

优化 Spark Plan

物理阶段的优化是从逻辑优化阶段输出的 Optimized Logical Plan 开始的，因此我们先来回顾一下小 Q 的原始查询和 Optimized Logical Plan。

复制代码

```
1
2  val userFile: String = _
3  val usersDf = spark.read.parquet(userFile)
4  usersDf.printSchema
5  /**
6  root
7  |-- userId: integer (nullable = true)
8  |-- name: string (nullable = true)
9  |-- age: integer (nullable = true)
10 |-- gender: string (nullable = true)
11 |-- email: string (nullable = true)
12 */
13 val users = usersDf
14 .select("name", "age", "userId")
15 .filter($"age" < 30)
16 .filter($"gender".isin("M"))
17
18 val txFile: String = _
19 val txDf = spark.read.parquet(txFile)
20 txDf.printSchema
21 /**
22 root
23 |-- txId: integer (nullable = true)
24 |-- userId: integer (nullable = true)
25 |-- price: float (nullable = true)
26 |-- volume: integer (nullable = true)
27 */
28
```

```
val result = txDF.select("price", "volume", "userId")
.join(users, Seq("userId"), "inner")
.groupBy(col("name"), col("age")).agg(sum(col("price") * col("volume")).alias(

result.write.parquet("_")

```

两表关联的查询语句经过转换之后，得到的 Optimized Logical Plan 如下图所示。注意，在逻辑计划的根节点，出现了“Join Inner”字样，Catalyst 优化器明确了这一步需要做内关联。但是，怎么做内关联，使用哪种 Join 策略来进行关联，Catalyst 并没有交代清楚。因此，逻辑计划本身不具备可操作性。

```
== Optimized Logical Plan ==
Join Inner, (userId#56 = userId#66)
:- Project [userId#56, name#57, age#58]
:  +- Filter ((((isnotnull(age#58) AND isnotnull(gender#59))
:             AND (age#58 < 30)) AND (gender#59 = M)) AND isnotnull(userId#56))
:     +- Relation[userId#56,name#57,age#58,gender#59,email#60] parquet
+- Filter isnotnull(userId#66)
   +- Relation[userId#66,itemId#67,price#68,quantity#69] parquet
```

小Q变身：Optimized Logical Plan

为了让查询计划（Query Plan）变得可操作、可执行，Catalyst 的物理优化阶段（Physical Planning）可以分为两个环节：优化 Spark Plan 和生成 Physical Plan。

> 在优化 Spark Plan 的过程中，Catalyst 基于既定的优化策略（Strategies），把逻辑计划中的关系操作符一一映射成物理操作符，生成 Spark Plan。
>
> 在生成 Physical Plan 过程中，Catalyst 再基于事先定义的 Preparation Rules，对 Spark Plan 做进一步的完善、生成可执行的 Physical Plan。

那么问题来了，在优化 Spark Plan 的过程中，Catalyst 都有哪些既定的优化策略呢？从数量上来说，Catalyst 有 14 类优化策略，其中有 6 类和流计算有关，剩下的 8 类适用于所有的计算场景，如批处理、数据分析、机器学习和图计算，当然也包括流计算。因此，我们只需了解这 8 类优化策略。

序号	类别	优化策略	含义&作用
1	通用	BasicOperators	逻辑到物理的基本映射：如Project、Filter、Sort等
2		JoinSelection	静态的Join策略选择
3		InMemoryScans	缓存策略，对应逻辑优化阶段的Cache Manager
4		Aggregation	聚合策略
5		Window	窗口计算策略
6		SpecialLimits	与Limit相关的优化策略
7		PythonEvals	Python UDF优化策略
8		SparkScripts	Transformation脚本优化策略
9	Streaming	StatefulAggregationStrategy	有状态的聚合策略
10		StreamingDeduplicationStrategy	流处理中的去重策略
11		StreamingGlobalLimitStrategy	流处理中的Limit处理策略
12		StreamingJoinStrategy	流处理中的Join策略
13		StreamingRelationStrategy	数据源读取策略
14		FlatMapGroupsWithStateStrategy	流处理中的FlatMap优化

极客时间

Catalyst物理优化阶段的14个优化策略

所有优化策略在转换方式上都大同小异，都是使用基于模式匹配的偏函数（Partial Functions），把逻辑计划中的操作符平行映射为 Spark Plan 中的物理算子。比如，BasicOperators 策略直接把 Project、Filter、Sort 等逻辑操作符平行地映射为物理操作符。其他策略的优化过程也类似，因此，在优化 Spark Plan 这一环节，咱们只要抓住一个“典型”策略，掌握它的转换过程即可。

那我们该抓谁做“典型”呢？我觉得，**这个“典型”至少要满足两个标准：一，它要在我们的应用场景中非常普遍；二，它的取舍对于执行性能的影响最为关键。**以这两个标准去遴选上面的 8 类策略，我们分分钟就能锁定 JoinSelection。接下来，我们就以 JoinSelection 为例，详细讲解这一环节的优化过程。

如果用一句话来概括 JoinSelection 的优化过程，就是结合多方面的信息，来决定在物理优化阶段采用哪种 Join 策略。那么问题来了，Catalyst 都有哪些 Join 策略？

Catalyst 都有哪些 Join 策略？

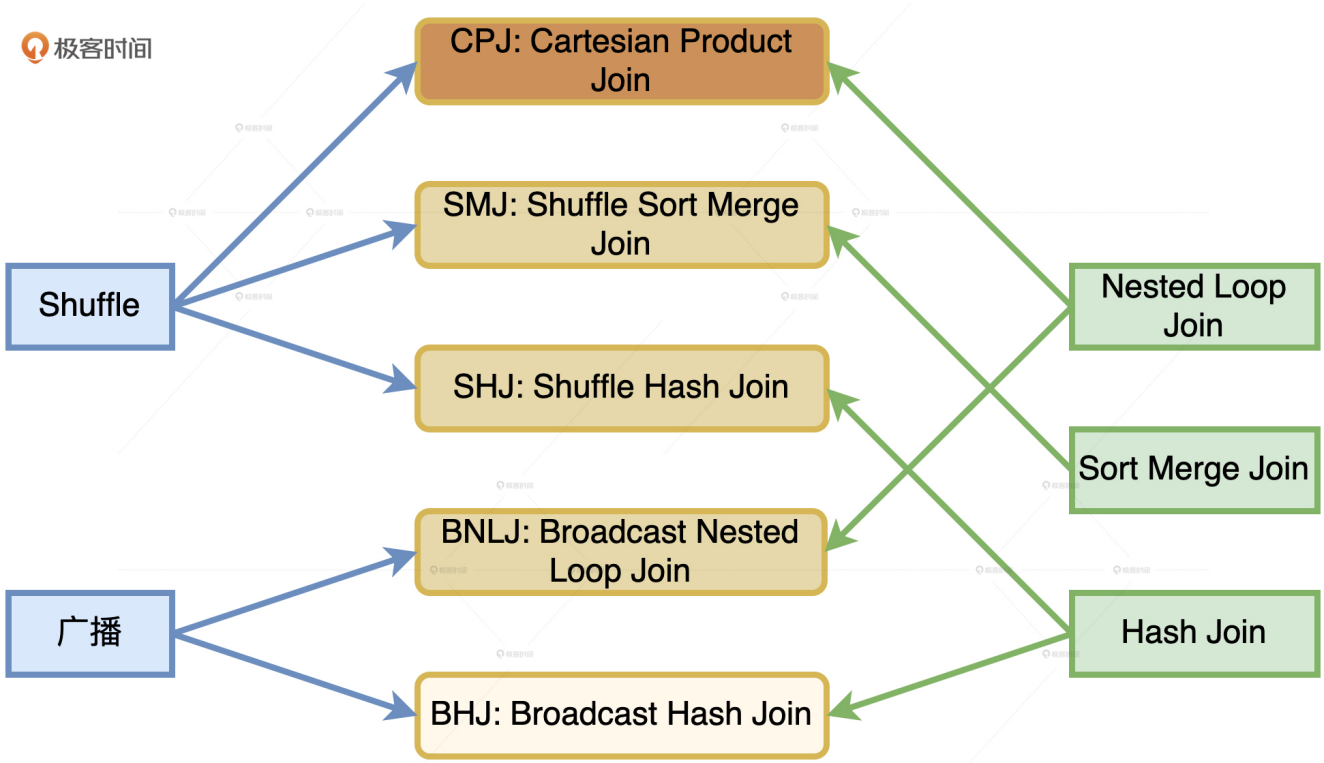
结合 Joins 的实现机制和数据的分发方式，Catalyst 在运行时总共支持 5 种 Join 策略，分别是 Broadcast Hash Join（BHJ）、Shuffle Sort Merge Join（SMJ）、Shuffle Hash Join（SHJ）、Broadcast Nested Loop Join（BNLJ）和 Shuffle Cartesian Product Join（CPJ）。

Join策略	执行效率排序	含义
BHJ	最优	在小表构建哈希表，通过把小表广播的方式完成关联
SMJ	次优	先Shuffle、再排序，用Sort Merge Join方式完成关联
SHJ		先Shuffle、再构建哈希表，用Hash Join方式完成关联
BNLJ	最差	将小表广播到所有Executors，以NLJ方式完成关联
CPJ		先Shuffle，然后用Nested Loop Join方式完成关联

极客时间

5种Join策略及其含义

通过上表中 5 种 Join 策略的含义，我们知道，它们是来自 2 种数据分发方式（广播和 Shuffle）与 3 种 Join 实现机制（Hash Joins、Sort Merge Joins 和 Nested Loop Joins）的排列组合。那么，在 JoinSelection 的优化过程中，Catalyst 会基于什么逻辑，优先选择哪种 Join 策略呢?



数据分发方式与Join实现机制的排列组合

JoinSelection 如何决定选择哪一种 Join 策略?

逻辑其实很简单，**Catalyst 总会尝试优先选择执行效率最高的策略。**具体来说，在选择 join 策略的时候，JoinSelection 会先判断当前查询是否满足 BHJ 所要求的先决条件：如果满足就立即选中 BHJ；如果不满足，就继续判断当前查询是否满足 SMJ 的先决条件。以此类推，直到最终选无可选，用 CPJ 来兜底。

那么问题来了，这 5 种 Join 策略都需要满足哪些先决条件呢？换句话说，JoinSelection 做决策时都要依赖哪些信息呢？

总的来说，**这些信息分为两大类，第一类是“条件型”信息，用来判决 5 大 Join 策略的先决条件。第二类是“指令型”信息，也就是开发者提供的 Join Hints。**

我们先来说“条件型”信息，它包含两种。第一种是 Join 类型，也就是是否等值、连接形式等，这种信息的来源是查询语句本身。第二种是内表尺寸，这些信息的来源就比较广泛了，可以是 Hive 表之上的 ANALYZE TABLE 语句，也可以是 Spark 对于 Parquet、ORC、CSV 等源文件的尺寸预估，甚至是来自 AQE 的动态统计信息。

5 大 Join 策略对于这些信息的要求，我都整理到了下面的表格里，你可以看一看。

Join策略	Join类型			表大小
	要求等值Join	必须是Inner Join	不能是Full Outer Join	要求内表可以放到广播变量
BHJ	R		R	R
SMJ	R			
SHJ	R			
BNLJ				R
CPJ		R		

极客时间

5种Join策略的先决条件

指令型信息也就是 Join Hints，它的种类非常丰富，它允许我们把个人意志凌驾于 Spark SQL 之上。比如说，如果我们对小 Q 的查询语句做了如下的调整，JoinSelection 在做 Join 策略选择的时候就会优先尊重我们的意愿，跳过 SMJ 去选择排序更低的 SHJ。具体的代码示例如下：

复制代码

```
val result = txDF.select("price", "volume", "userId")
.join(users.hint("shuffle_hash"), Seq("userId"), "inner")
.groupBy(col("name"), col("age")).agg(sum(col("price") *
col("volume")).alias("revenue"))
```

熟悉了 JoinSelection 选择 Join 策略的逻辑之后，我们再来看小 Q 是怎么选择的。小 Q 是典型的星型查询，也就是事实表与维度表之间的数据关联，其中维表还带过滤条件。在决定采用哪种 Join 策略的时候，JoinSelection 优先尝试判断小 Q 是否满足 BHJ 的先决条件。

显然，小 Q 是等值的 Inner Join，因此表格中 BHJ 那一行的前两个条件小 Q 都满足。但是，内表 users 尺寸较大，超出了广播阈值的默认值 10MB，不满足 BHJ 的第三个条件。因此，JoinSelection 不得不忍痛割爱、放弃 BHJ 策略，只好退而求其次，沿着表格继续向下，尝试判断小 Q 是否满足 SMJ 的先决条件。

SMJ 的先决条件很宽松，查询语句只要是等值 Join 就可以。小 Q 自然是满足这个条件的，因此 JoinSelection 最终给小 Q 选定的 Join 策略就是 SMJ。下图是小 Q 优化过后的 Spark Plan，从中我们可以看到，查询计划的根节点正是 SMJ。

```
//SparkPlan
SortMergeJoin [userId#56], [userId#66], Inner
:- Project [userId#56, name#57, age#58]
:  +- Filter ((((isnotnull(age#58) AND isnotnull(gender#59)) AND (age#58 < 30)) AND (gender#59 = M)) AND isnotnull(userId#56))
:     +- FileScan parquet [userId#56,name#57,age#58,gender#59] Batched: true,
        DataFilters: [isnotnull(age#58), isnotnull(gender#59), (age#58 < 30), (gender#59 = M), isnotnull(userId#56)],
        Format: Parquet, Location: InMemoryFileIndex[file:/tmp/users.parquet], PartitionFilters: [],
        PushedFilters: [IsNotNull(age), IsNotNull(gender), LessThan(age,30), EqualTo(gender,M), IsNotNull(userId)],
        ReadSchema: struct<userId:int,name:string,age:int,gender:string>
+- Project [userId#66, itemId#67, price#68, quantity#69]
   +- Filter isnotnull(userId#66)
      +- FileScan parquet [userId#66,itemId#67,price#68,quantity#69] Batched: true,
         DataFilters: [isnotnull(userId#66)], Format: Parquet,
         Location: InMemoryFileIndex[file:/tmp/transactions.parquet], PartitionFilters: [],
         PushedFilters: [IsNotNull(userId)], ReadSchema: struct<userId:int,itemId:int,price:double,quantity:int>
```

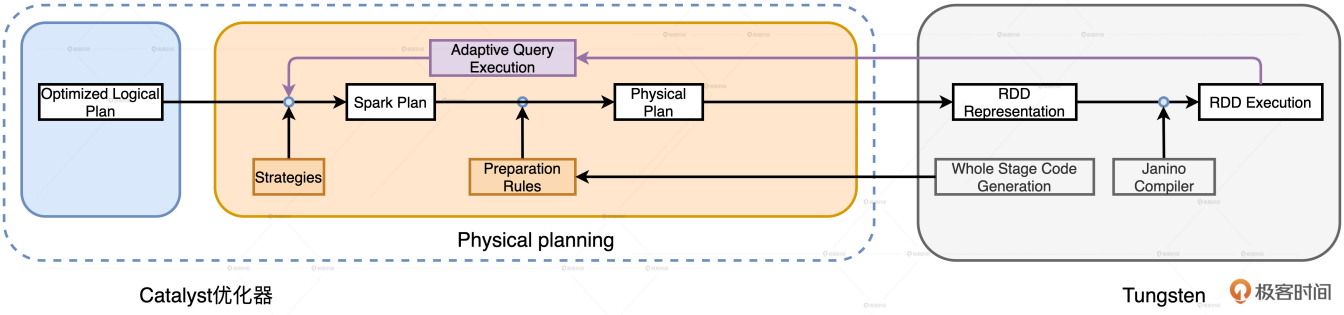
小Q再变身：Spark Plan

现在我们知道了 Catalyst 都有哪些 Join 策略，JoinSelection 如何对不同的 Join 策略做选择。小 Q 也从 Optimized Logical Plan 摇身一变，转换成了 Spark Plan，也明确了在运行时采用 SMJ 来做关联计算。不过，即使小 Q 在 Spark Plan 中已经明确了每一步该“怎么做”，但是，Spark 还是做不到把这样的查询计划转化成可执行的分布式任务，这又是为什么呢？

生成 Physical Plan

原来，Shuffle Sort Merge Join 的计算需要两个先决条件：Shuffle 和排序。而 Spark Plan 中并没有明确指定以哪个字段为基准进行 Shuffle，以及按照哪个字段去做排序。

因此，Catalyst 需要对 Spark Plan 做进一步的转换，生成可操作、可执行的 Physical Plan。具体怎么做呢？我们结合 Catalyst 物理优化阶段的流程图来详细讲讲。



物理计划阶段

从上图中我们可以看到，从 Spark Plan 到 Physical Plan 的转换，需要几组叫做 Preparation Rules 的规则。这些规则坚守最后一班岗，负责生成 Physical Plan。那么，这些规则都是什么，它们都做了哪些事情呢？我们一起来看一下。

Preparation Rules	含义	作用
EnsureRequirements	确保每个操作符的输入要求，必要时添加Shuffle、Sort等操作	为Physical Plan补充必要的操作步骤，保证Spark Plan中计划的每一个步骤能够顺利执行
CollapseCodegenStages	Tungsten优化机制：全阶段代码生成（Whole Stage Code Generation）	在同一个Stage内部，尽可能地把所有操作和计算捏合成一个函数，提升计算效率
ReuseExchange	内存或是磁盘中的存储复用	同样的执行计划可以共享广播变量或是Shuffle的中间结果，避免重复的Shuffle操作
ReuseSubquery	子查询复用	复用同样的查询结果，避免重复计算
PlanSubquery	生成子查询	对子查询应用Preparation Rules
ExtractPythonUDFs	提取Python的UDF函数	把Python UDF分发到单独的Python进程

极客时间

Preparation Rules

Preparation Rules 有 6 组规则，这些规则作用到 Spark Plan 之上就会生成 Physical Plan，而 Physical Plan 最终会由 Tungsten 转化为用于计算 RDD 的分布式任务。

小 Q 的查询语句很典型，也很简单，并不涉及子查询，更不存在 Python UDF。因此，在小 Q 的例子中，我们并不会用到子查询、数据复用或是 Python UDF 之类的规则，只有 EnsureRequirements 和 CollapseCodegenStages 这两组规则会用到小 Q 的 Physical Plan 转化中。

实际上，它们也是结构化查询中最常见、最常用的两组规则。今天，我们先来重点说说 EnsureRequirements 规则的含义和作用。至于 CollapseCodegenStages 规则，它实际上就是 Tungsten 的 WSCG 功能，我们下一讲再详细说。

EnsureRequirements 规则

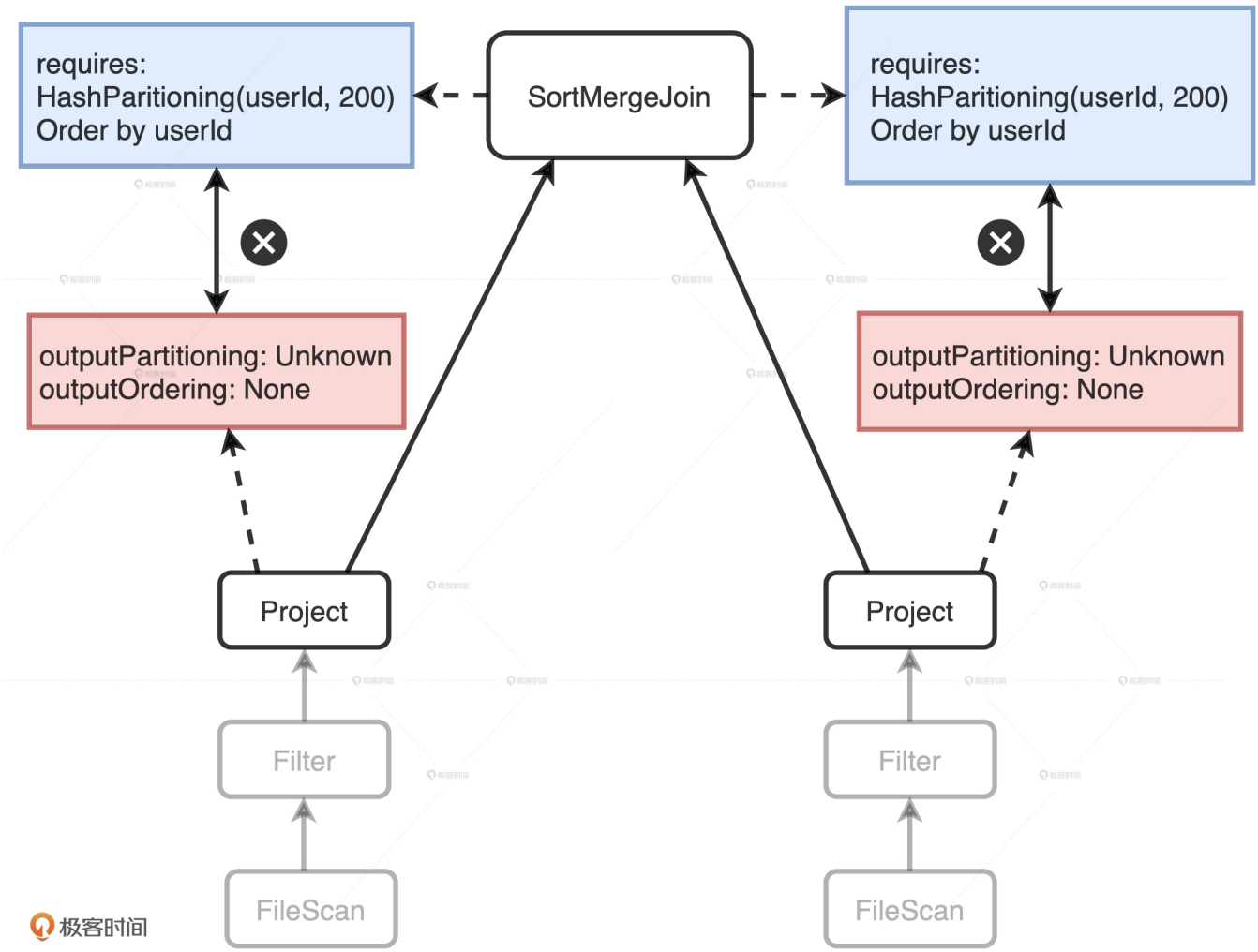
EnsureRequirements 翻译过来就是“确保满足前提条件”，这是什么意思呢？对于执行计划中的每一个操作符节点，都有 4 个属性用来分别描述数据输入和输出的分布状态。

操作符属性	含义
outputPartitioning	输出数据的分区规则
outputOrdering	输出数据的排序规则
requireChildDistribution	要求输入数据满足某种分区规则
requireChildOrdering	要求输入数据满足某种排序规则

极客时间

描述输入、输出要求的4个属性

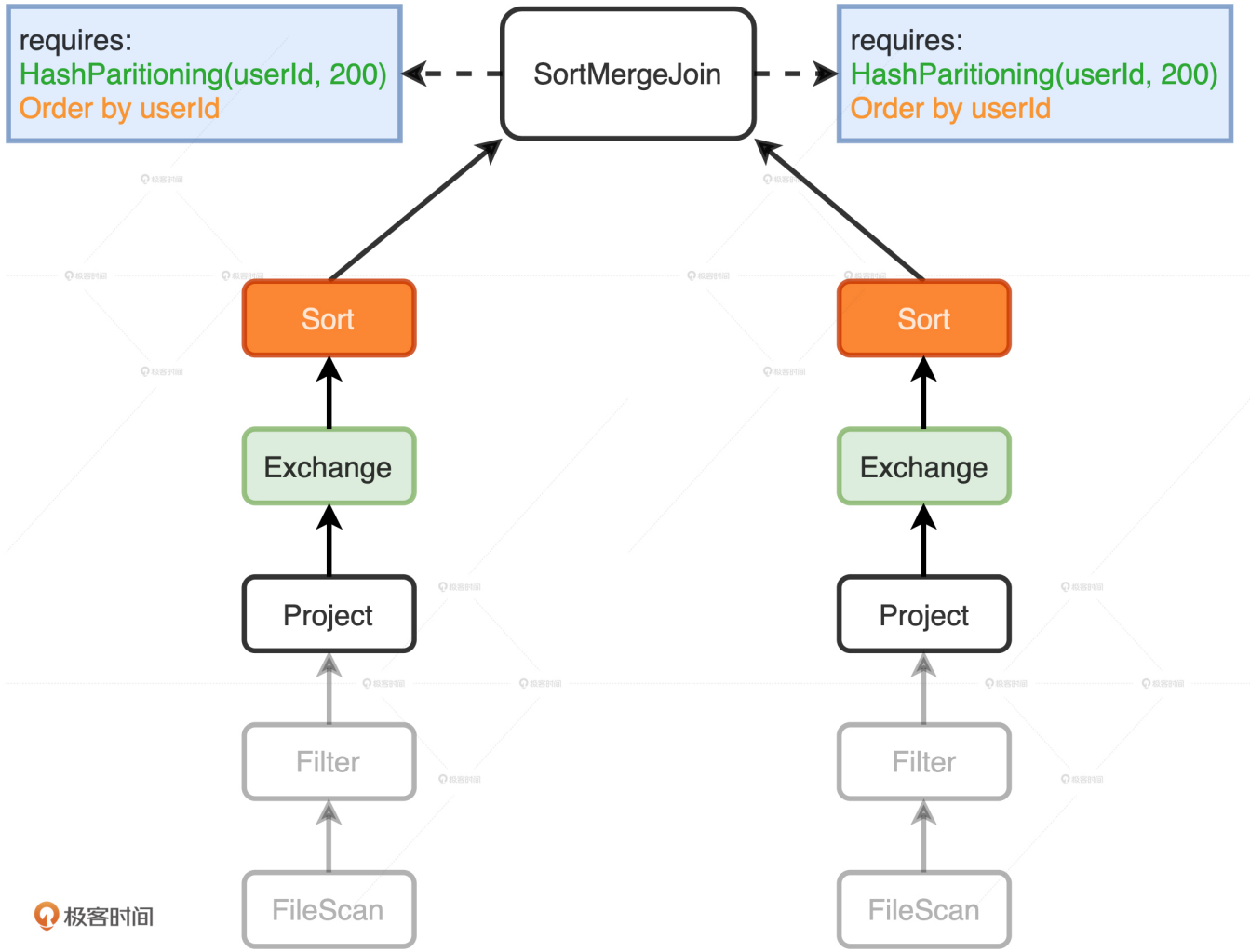
EnsureRequirements 规则要求，子节点的输出数据要满足父节点的输入要求。这又怎么理解呢？



Project没有满足SortMergeJoin的Requirements

我们以小 Q 的 Spark Plan 树形结构图为例，可以看到：图中左右两个分支分别表示扫描和处理 users 表和 transactions 表。在树的最顶端，根节点 SortMergeJoin 有两个 Project 子节点，它们分别用来表示 users 表和 transactions 表上的投影数据。这两个 Project 的 outputPartitioning 属性和 outputOrdering 属性分别是 Unknow 和 None。因此，它们输出的数据没有按照任何列进行 Shuffle 或是排序。

但是，SortMergeJoin 对于输入数据的要求很明确：按照 userId 分成 200 个分区且排好序，而这两个 Project 子节点的输出显然并没有满足父节点 SortMergeJoin 的要求。这个时候，**EnsureRequirements 规则就要介入了，它通过添加必要的操作符，如 Shuffle 和排序，来保证 SortMergeJoin 节点对于输入数据的要求一定要得到满足**，如下图所示。



EnsureRequirements规则添加Exchange和Sort操作

在两个 Project 节点之后，EnsureRequirements 规则分别添加了 Exchange 和 Sort 节点。其中 Exchange 节点代表 Shuffle 操作，用来满足 SortMergeJoin 对于数据分布的要求；Sort 表示排序，用于满足 SortMergeJoin 对于数据有序的要求。

添加了必需的节点之后，小 Q 的 Physical Plan 已经相当具体了。这个时候，Spark 可以通过调用 Physical Plan 的 doExecute 方法，把结构化查询的计算结果，转换成 RDD[InternalRow]，这里的 InternalRow，就是 Tungsten 设计的定制化二进制数据结构，这个结构我们在内存视角（一）有过详细的讲解，你可以翻回去看看。通过调用 RDD[InternalRow]之上的 Action 算子，Spark 就可以触发 Physical Plan 从头至尾依序执行。

最后，我们再来看看小 Q 又发生了哪些变化。

```
== Physical Plan ==
*(5) SortMergeJoin [userId#56], [userId#66], Inner
:- *(2) Sort [userId#56 ASC NULLS FIRST], false, 0
:  +- Exchange hashpartitioning(userId#56, 200), true, [id=#55]
:     +- *(1) Project [userId#56, name#57, age#58]
:        +- *(1) Filter ((((isnotnull(age#58) AND isnotnull(gender#59))
                        AND (age#58 < 30)) AND (gender#59 = M)) AND isnotnull(userId#56))
:           +- *(1) ColumnarToRow
:              +- FileScan parquet [userId#56,name#57,age#58,gender#59] Batched: true,
                 DataFilters: [isnotnull(age#58), isnotnull(gender#59), (age#58 < 30), (gender#59 = M), isnotnull(userId#56)],
                 Format: Parquet, Location: InMemoryFileIndex[file:/tmp/users.parquet], PartitionFilters: [],
                 PushedFilters: [IsNotNull(age), IsNotNull(gender), LessThan(age,30), EqualTo(gender,M), IsNotNull(userId)],
                 ReadSchema: struct<userId:int,name:string,age:int,gender:string>
+- *(4) Sort [userId#66 ASC NULLS FIRST], false, 0
   +- Exchange hashpartitioning(userId#66, 200), true, [id=#65]
      +- *(3) Project [userId#66, itemId#67, price#68, quantity#69]
         +- *(3) Filter isnotnull(userId#66)
            +- *(3) ColumnarToRow
               +- FileScan parquet [userId#66,itemId#67,price#68,quantity#69] Batched: true,
                  DataFilters: [isnotnull(userId#66)], Format: Parquet,
                  Location: InMemoryFileIndex[file:/tmp/transactions.parquet], PartitionFilters: [],
                  PushedFilters: [IsNotNull(userId)], ReadSchema: struct<userId:int,itemId:int,price:double,quantity:int>
```

小Q再变身：Physical Plan

首先，我们看到 EnsureRequirements 规则在两个分支的顶端分别添加了 Exchange 和 Sort 操作，来满足根节点 SortMergeJoin 的计算需要。其次，如果你仔细观察的话，会发现 Physical Plan 中多了很多星号“*”，这些星号的后面还带着括号和数字，如图中的“*（3）”、“*（1）”。这种星号“*”标记表示的就是 WSCG，后面的数字代表 Stage 编号。因此，括号中数字相同的操，最终都会被捏合成一份“手写代码”，也就是我们下一讲要说的 Tungsten 的 WSCG。

至此，小 Q 从一个不考虑执行效率的“叛逆少年”，就成长为了一名执行高效的“专业人士”，Catalyst 这位人生导师在其中的作用功不可没。

小结

为了把逻辑计划转换为可以交付执行的物理计划，Spark SQL 物理优化阶段包含两个环节：优化 Spark Plan 和生成 Physical Plan。

在优化 Spark Plan 这个环节，Catalyst 基于既定的策略把逻辑计划平行映射为 Spark Plan。策略很多，我们重点掌握 JoinSelection 策略就可以，它被用来在运行时选择最佳的 Join 策略。JoinSelection 按照 BHJ > SMJ > SHJ > BNLJ > CPJ 的顺序，依次判断查询语句是否满足每一种 Join 策略的先决条件进行“择优录取”。

如果开发者不满足于 JoinSelection 默认的选择顺序，也就是 BHJ > SMJ > SHJ > BNLJ > CPJ，还可以通过在 SQL 或是 DSL 语句中引入 Join hints，来明确地指定 Join 策略，

从而把自己的意愿凌驾于 Catalyst 之上。不过，需要我们注意的是，要想让指定的 Join 策略在运行时生效，查询语句也必须要满足其先决条件才行。

在生成 Physical Plan 这个环节，Catalyst 基于既定的几组 Preparation Rules，把优化过后的 Spark Plan 转换成可以交付执行的物理计划，也就是 Physical Plan。在这些既定的 Preparation Rules 当中，你需要重点掌握 EnsureRequirements 规则。

EnsureRequirements 用来确保每一个操作符的输入条件都能够得到满足，在必要的时候，会把必需的操作符强行插入到 Physical Plan 中。比如对于 Shuffle Sort Merge Join 来说，这个操作符对于子节点的数据分布和顺序都是有明确要求的，因此，在子节点之上，EnsureRequirements 会引入新的操作符如 Exchange 和 Sort。

每日一练

3 种 Join 实现方式和 2 种网络分发模式，明明应该有 6 种 Join 策略，为什么 Catalyst 没有支持 Broadcast Sort Merge Join 策略?

期待在留言区看到你的思考和答案，我们下一讲见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 21 | Catalyst逻辑计划：你的SQL语句是怎么被优化的？（上）

下一篇 23 | 钨丝计划：Tungsten给开发者带来了哪些福报？

精选留言 (6)

写留言

kingcall
2021-05-03

回答问题：
个人认为没有必要：因为被广播出去的数据集合都很小，可以通过Hash 或者 Nested Loop 来实现，而这二者之间的区别在于
1. Hash 主要适用于等值关联
2. Nested Loop 就可以用来实现笛卡尔积或者是不等值关联...
展开

作者回复: 标准答案了，满分💯~

关于SMJ和CPJ的分析也很到位，对于非等值Join，如果非要用SMJ来实现行不行？其实也不是不可以。

但如果细想的话，你会发现，对于非等值Join来说，即便你打算尝试用SMJ去实现，最终实现下来你发现算法的复杂度，和你使用Nested Loop Join双层for循环的算法是等价的。而SMJ还有额外的排序开销，因此里外里算下来，还不如直接用NLJ也就是CPJ来实现。

1 2

aof
2021-05-05

broadcast广播模式下，排序没啥必要吧，因为本身被广播的数据集就比较小，hash join和NLJ完全够用了。而且SMJ本身就是针对大表关联大表设计的join算法

展开

作者回复: 正解，满分 💯 ~

zxk
2021-05-04

没必要实现 BSMJ， Shuffle 排序的目的是为了将同个 partition 的数据汇集到一起发送，而 Broadcast 本身就是对数据的全量发送，并不需要区分数据归属于哪个 partition。

提问：老师说 Tungsten 的 WSCG 会将同个 stage 内的操作捏合成一个手写代码，意思是指你合成一个函数操作么？如果是，那么跟 DAGScheduler 中的函数合并有什么区别...

展开

作者回复: 第一题可以参考kingcall同学的答案哈~

后面的提问是个好问题。逻辑优化、物理优化、Tungsten优化，是Spark SQL智能大脑的“三招一套”，这3部分就是在讲，一份代码（DataFrame、Dataset、SQL）是如何一步步被优化、最终被融合成一份“手写代码”，然后交付DAGScheduler做分布式执行。在接下来的23讲，咱们重点关注Tungsten优化环节，也会着重解释“手写代码”和“RDD迭代器嵌套”之间的本质区别，敬请期待哈~

GEEKBANG_2743065
2021-05-04

Sort Merger Join适用于大表，大表无法broadcast，所以没有broadcast sort merge joi

n。

展开

作者回复: 超出广播阈值的大表，确实无法广播，不过，咱们的问题是，对于可以广播的小表，为什么Spark社区没有将Broadcast和SMJ结合到一起~

可以参考Kingcall同学的答案哈~

Fendora范东_

2021-05-03

一、从原理和复杂度看(假设单节点大表规模m,小表规模n)
1、数据分发:
broadcast把整个小表发送到大表数据所在节点;
shuffle大小表按照同样的分区方式、数进行数据重新 划分。
2、join实现...

展开

作者回复: 第一个，不同Join实现机制的算法复杂度分析得没问题~

第二个，两种实现方法的复杂度计算也没问题~

最大的问题是，题没读懂~ [允悲] 咱们是想问：为什么Spark SQL在6种Join策略中，只支持本讲中提到的5种，而没有实现Broadcast SMJ?

Z宇锤锤

2021-05-03

Broadcast Sort Merger Join中Broadcast指的是数据分发的方式，SMB指的是Join实现机制。
Sort Merge Join的原理是将两张表的数据按照相同的分区算法，分发到各个Executor上。如果使用Broadcast传输，被广播的表会先在Executor端进行数据的拆分，拆分完成以后，所有的分区会被Collect到Driver端，再向每一个Executor分发完整数据。这使被...

展开

作者回复: 前面说的没错，Broadcast是数据分发方式，SMJ是Join实现机制。

不过，后面的不太对。Spark社区之所以没有把Broadcast和SMJ结合在一起，其实核心原因很简

单：没必要。

因为，在Broadcast的机制下，每个Executors都有维表的全量数据，且维表足够小，这个时候，使用Hash Join的实现机制，远比SMJ的机制的执行效率更高。换句话说，在这种情况下，Hash Join不存在稳定性问题，而它的性能又更好，因此，Spark社区没有必要再去实现SMJ的实现机制。